Parallel Computing Final Term: *Mean Shift*

Giovanni Bindi - 7016072
giovanni.bindi@stud.unifi.it
https://github.com/w00zie/mean_shift

Abstract

In this project I've developed three implementations of the mean shift clustering algorithm. These implementations are one sequential version and two parallel variants, obtained with OpenMP and CUDA. A simple performance study has then been operated, in order to analyze and quantify the benefits of parallelism. The developed code comes (almost in its entirety) as an header-only C++17 library, with simple test cases to verify the correct behaviour of the implementation on a synthetic data-set.

Future Distribution Permission

1. Brief Introduction

The mean shift clustering algorithm is a popular non-parametric clustering technique, developed in its first form during the 1970s [3]. It aims to locate the the *modes* of a density function, given discrete data sampled from that function. This is done with the use of a kernel, hence the mean shift procedure operates a kernel density estimation. Formally, given a set of observations $\mathcal{S} = \{\mathbf{x}_1, \dots, \mathbf{x}_N\}$, where $\mathbf{x}_i \in \mathbb{R}^D$, the kernel density estimate obtained with a kernel $K(\cdot)$ and and window radius h is

$$f(\mathbf{x}) = \frac{1}{Nh^D} \sum_{i=1}^{N} K\Big(\frac{\mathbf{x} - \mathbf{x}_i}{h}\Big) \tag{1}$$

For radially symmetric kernels, it suffices to define the profile of the kernel $k(\cdot)$ satisfying

$$K(\mathbf{x}) = ck(\mathbf{x}) \tag{2}$$

where c is a normalization constant that assures $K(\mathbf{x})$ integrates to 1. The modes of the density function are located at the zeros of the gradient $\nabla f(\mathbf{x})$. From this principle it is possible to derive an iterative algorithm that is able to perform clustering [1].

2. Algorithm

Starting from the initial estimate $\mathbf{x}$ and the choice of the kernel function $K(\cdot)$, the weighted mean of the density, determined by K is

$$m(\mathbf{x}) = \frac{\sum_{\mathbf{x}_i \in N(\mathbf{x})} K(\mathbf{x} - \mathbf{x}_i)\mathbf{x}_i}{\sum_{\mathbf{x}_i \in N(\mathbf{x})} K(\mathbf{x} - \mathbf{x}_i)} \tag{3}$$

where $N(\mathbf{x})$ is a neighborhood of $\mathbf{x}$. Typically a gaussian kernel on the distance to the current estimate is used

$$K_\sigma(\mathbf{x} - \mathbf{x}_i) = e^{\frac{\|\mathbf{x} - \mathbf{x}_i\|}{2\sigma^2}} \tag{4}$$

where the hyperparameter σ^2 plays the role of the window radius h. The difference $m(\mathbf{x}) - \mathbf{x}$ is called *mean shift*. The algorithm proceeds as following, being I the total number of iterations and $\mathcal{S}^{(t)} = \{\mathbf{x}_1^{(t)}, \dots, \mathbf{x}_N^{(t)}\}$, for $t = 1, \dots, I$:

1. **Calculate** $m(\mathbf{x}^{(t)})$, for each $\mathbf{x}^{(t)} \in \mathcal{S}^{(t)}$
2. **Update** $\mathbf{x}^{(t+1)} = m(\mathbf{x}^{(t)})$, for each $\mathbf{x}^{(t)} \in \mathcal{S}^{(t)}$

Where $\mathcal{S}^{(1)} = \mathcal{S}$ is the dataset.
A possible stopping criterion is to check for convergence, *i.e.* when $\|m(\mathbf{x}^{(t)}) - \mathbf{x}^{(t)}\| \le \epsilon$ for each $\mathbf{x}^{(t)} \in \mathcal{S}$. This project's implementation can be described by the pseudo-code in Algorithm 1.

Algorithm 1 Sequential *Mean Shift*

```
Input: $\mathcal{S} \subset \mathbb{R}^D$. $I \in \mathbb{N}$. $\sigma, R, \delta \in \mathbb{R}$.
Input: Optional: $\epsilon \in \mathbb{R}$.
Output: Centroids $\{\boldsymbol{\mu}_1, \dots \boldsymbol{\mu}_k\}$ for some $k \in \mathbb{N}$
 1: for $t = 1, \dots, I$ do
 2:   for $i = 1, \dots, N$ do
 3:     $\boldsymbol{\tau} = \mathbf{0}$
 4:     $\eta = 0$
 5:     for $j = 1, \dots, N$ do
 6:       if $\|\mathbf{x}_i^{(t)} - \mathbf{x}_j^{(t)}\|^2 \leq R$ then
 7:         $\boldsymbol{\tau} = \boldsymbol{\tau} + K_\sigma(\mathbf{x}_i^{(t)} - \mathbf{x}_j^{(t)})\mathbf{x}_j^{(t)}$
 8:         $\eta = \eta + K_\sigma(\mathbf{x}_i^{(t)} - \mathbf{x}_j^{(t)})$
 9:       end if
10:     end for
11:     $\mathbf{x}_i^{(t+1)} = \boldsymbol{\tau}/\eta$
12:     if $\|\mathbf{x}_i^{(t+1)} - \mathbf{x}_i^{(t)}\| \leq \epsilon$ then
13:       stop_shifting($\mathbf{x}_i^{(t+1)}$)
14:     end if
15:   end for
16: end for
17: $\{\boldsymbol{\mu}_1, \dots \boldsymbol{\mu}_k\} \leftarrow$ reduce_centroids($\mathcal{S}^{(I)}, \delta$)
18: return $\{\boldsymbol{\mu}_1, \dots \boldsymbol{\mu}_k\}$
```

The method `stop_shifting()` sets a boolean flag (not shown in the algorithm for brevity) that prevents a point to be shifted during the future iterations. This is used only if an ϵ-based stopping criterion is desired.

The method `reduce_centroids()` reduces the converged data points to a subset that, ideally, should be the cluster centers. This is done with the help of a variable, $\delta \in \mathbb{R}$, that has to be set by the user. This quantity represents the minimum distance for which two points can be considered belonging to the same cluster.

The time complexity of this algorithm is $O(IN^2)$.

3. OpenMP

OpenMP is an API implementing multithreading in a fork-join model [2]. It consists of a set of compiler directives, library routines, and environment variables that influence the runtime behavior of the application: a primary thread forks a specified number of sub-threads and the system divides a task among them. The OpenMP interface is very simple and flexible, it just takes a simple directive to parallelize a portion of code: in Algorithm 2 blue text represents code that is executed in parallel.

Algorithm 2 OpenMP *Mean Shift*

```
Input: $\mathcal{S} \subset \mathbb{R}^D$. $I \in \mathbb{N}$. $\sigma, R, \delta \in \mathbb{R}$.
Input: Optional: number of threads $T \in \mathbb{N}$.
Input: Optional: $\epsilon \in \mathbb{R}$.
Output: Centroids $\{\boldsymbol{\mu}_1, \dots \boldsymbol{\mu}_k\}$ for some $k \in \mathbb{N}$
 1: for $t = 1, \dots, I$ do
 2:   #pragma omp parallel for (dynamic or $T$)
 3:   for $i = 1, \dots, N$ do
 4:     $\boldsymbol{\tau} = \mathbf{0}$
 5:     $\eta = 0$
 6:     for $j = 1, \dots, N$ do
 7:       if $\|\mathbf{x}_i^{(t)} - \mathbf{x}_j^{(t)}\|^2 \leq R$ then
 8:         $\boldsymbol{\tau} = \boldsymbol{\tau} + K_\sigma(\mathbf{x}_i^{(t)} - \mathbf{x}_j^{(t)})\mathbf{x}_j^{(t)}$
 9:         $\eta = \eta + K_\sigma(\mathbf{x}_i^{(t)} - \mathbf{x}_j^{(t)})$
10:       end if
11:     end for
12:     #pragma omp critical
13:     $\mathbf{x}_i^{(t+1)} = \boldsymbol{\tau}/\eta$
14:     if $\|\mathbf{x}_i^{(t+1)} - \mathbf{x}_i^{(t)}\| \leq \epsilon$ then
15:       #pragma omp critical
16:       stop_shifting($\mathbf{x}_i^{(t+1)}$)
17:     end if
18:   end for
19: end for
20: $\{\boldsymbol{\mu}_1, \dots \boldsymbol{\mu}_k\} \leftarrow$ reduce_centroids($\mathcal{S}^{(I)}, \delta$)
21: return $\{\boldsymbol{\mu}_1, \dots \boldsymbol{\mu}_k\}$
```

Directive at line 2 parallelizes the subsequent for-loop: the number of spawned threads can be decided at runtime by OpenMP (`dynamic`), or it can be set by the user by specifying T. In this project both these two cases have been experimented.

The directive `#pragma omp critical` signals OpenMP that the following portion of the code is a critical section. This mechanism is employed since different threads write to the same shared variable, hence possibly causing a race condition. Ideally in this case, since different threads write to different memory location, this barrier should not be needed. Nevertheless this protection mechanism has been kept as a *good practice*.

4. CUDA

CUDA is a parallel computing platform and API model created by NVIDIA [4]. It allows the development of general purpose software on GPUs: the CUDA platform is a software layer that gives direct access to the GPU's instruction set and parallel computational elements, for the execution of compute *kernels*. A kernel is executed in parallel by an array of *threads*:

- All threads run the same code.
- Each thread has an ID that it uses to compute memory addresses and make control decisions.

Threads are arranged as a *grid* of thread *blocks*:

- Different kernels can have different grid/block configuration.
- Threads from the same block have access to a shared memory and their execution can be synchronized.

In this project two versions of the CUDA code have been produced: a *naive* version and a *tiled* one. The former is the simple "transcription" of the sequential algorithm to CUDA, while the latter takes advantage of the shared memory between the threads, ideally speeding up the calculations.

Data points have been stored as a $N \times D$ array, in order to operate coalesced memory accesses. A $1-$dimensional `blockDim` has then beed used.

The general structure for both implementations can be described by the pseudo-code in Algorithm 3. Variable T represents the number of threads per block. The total number of blocks per grid is then computed as $\lceil N/T \rceil$ at line 1.

The call `cudaDeviceSynchronize()` forces the program to wait for all previously issued commands (in this case the execution of one kernel) in all streams on the device to finish before continuing.

Again, blue text represents code that is executed in parallel. The unspecialized kernel call (`mean_shift()`, line 3) of Algorithm 3 represents both the *naive* and *tiled* implementations,

Algorithm 3 CUDA *Mean Shift*

```
Input: $\mathcal{S} \subset \mathbb{R}^D$. $I \in \mathbb{N}$. $\sigma, R, \delta \in \mathbb{R}$.
Input: Threads per block: $T \in \mathbb{N}$.
Output: Centroids $\{\boldsymbol{\mu}_1, \dots \boldsymbol{\mu}_k\}$ for some $k \in \mathbb{N}$
1: $B = \lceil N/T \rceil$
2: for $t = 1, \dots, I$ do
3:     mean_shift«B,T»($\mathcal{S}^{(t)}, \sigma, R$)
4:     cudaDeviceSynchronize()
5: end for
6: $\{\boldsymbol{\mu}_1, \dots \boldsymbol{\mu}_k\} \leftarrow$ reduce_centroids($\mathcal{S}^{(I)}, \delta$)
7: return $\{\boldsymbol{\mu}_1, \dots \boldsymbol{\mu}_k\}$
```

which are described in the next sections. Implementation details have been omitted from the descriptions of the two kernels, favoring a more abstract and high-level interpretation of the behaviour of the algorithms, as done with the other algorithm presented.

4.1. Naive

The naive version of the algorithm extracts the thread id (`tid` in the code) and computes the update from point $\mathbf{x}_{\mathtt{tid}}^{(t)}$ to $\mathbf{x}_{\mathtt{tid}}^{(t+1)}$. Before doing so, it checks if the thread is actually accessing an existing portion of the data.

Algorithm 4 CUDA *naive* kernel

```
Input: $\mathcal{S}^{(t)} \subset \mathbb{R}^D$. $\sigma, R \in \mathbb{R}$.
1: tid = (blockIdx.x × blockDim.x) + threadIdx.x
2: if tid < N then
3:     $\boldsymbol{\tau} = \mathbf{0}$
4:     $\eta = 0$
5:     for $i = 1, \dots, N$ do
6:         if $\|\mathbf{x}_{\mathtt{tid}}^{(t)} - \mathbf{x}_i^{(t)}\|^2 \leq R$ then
7:             $\boldsymbol{\tau} = \boldsymbol{\tau} + K_\sigma(\mathbf{x}_{\mathtt{tid}}^{(t)} - \mathbf{x}_i^{(t)})\mathbf{x}_i^{(t)}$
8:             $\eta = \eta + K_\sigma(\mathbf{x}_{\mathtt{tid}}^{(t)} - \mathbf{x}_i^{(t)})$
9:         end if
10:    end for
11:    $\mathbf{x}_{\mathtt{tid}}^{(t+1)} = \boldsymbol{\tau}/\eta$
12: end if
```

Memory accesses in the naive version are performed on the global memory. In CUDA, blocks are able to take advantage of a small private memory, whose content is shared between all the threads within a block. Code for this kernel can be seen in Appendix B.

4.2. Shared Memory

An optimization strategy is then operated by slicing the data into *tiles* and loading them sequentially into shared memory. The accesses are then performed on this local cache, amortizing the (more expensive) cost of accessing global memory. A common strategy is to have the size of these tiles being equal to the number of threads in a block, creating a one-to-one relationship between workers and memory addresses.

In this project T represents the number of threads (either CPU or GPU), hence it will be also used for sizing the shared memory. This is done at line 3 of Algorithm 5, where the shared memory $\mathcal{M}$ is initialized with a size of T (in particular as an array of size $T \times D$).

In order to fit the whole data into tiles of shared memory the algorithm proceeds iteratively, loading (at most) $\lceil N/T \rceil$ tiles of size T from global memory to the shared memory. Since T could not divide N evenly, there is the possibility of loading invalid data from the global memory. Another portion of shared memory, $\mathcal{V}$, is then used to store scalar "multipliers" (*i.e.* 0 or 1) that later in the code will help differentiating valid and invalid data.

For each tile τ an "inbound" check is operated and, consequently, valid data is loaded into shared memory. If this inbound test is not passed, local data is set to 0. This is expressed by the code from line 5 to 13.

Finally a synchronization barrier (`__syncthreads()`) is then used in order to wait the completion of this loading phase.

After this operation each thread loads the data (and the relative multipliers) from shared memory and proceeds to the computation of the *shifts*. In case of dealing with invalid data both the $\boldsymbol{\tau}$ and η variables will not be incremented, as written in lines 21 and 22.

After the completion of this process the same "inbound check" for `tid` is performed and the update is computed.

Code for this kernel can be seen in Appendix C.

Algorithm 5 CUDA *tiled* kernel

```
Input: $\mathcal{S}^{(t)} \subset \mathbb{R}^D$. $\sigma, R, \in \mathbb{R}$.
 1: $\omega =$ threadIdx.x
 2: tid = (blockIdx.x × blockDim.x) + $\omega$
 3: $\mathcal{M} \leftarrow$ shared array[$T \times D$]
 4: $\mathcal{V} \leftarrow$ shared array[$T$]
 5: $\boldsymbol{\tau} = \mathbf{0}$
 6: $\eta = 0$
 7: for $\tau = 1, \dots, \lceil N/T \rceil$ do
 8:     $\gamma = \tau \times T + \omega$
 9:     if $\gamma < N$ then
10:         $\mathcal{M}_\omega^{(\tau)} = \mathbf{x}_\gamma^{(t)}$  //Load data
11:         $\mathcal{V}_\omega^{(\tau)} = 1$  //Set valid data multiplier
12:     else
13:         $\mathcal{M}_\omega^{(\tau)} = \mathbf{0}$
14:         $\mathcal{V}_\omega^{(\tau)} = 0$
15:     end if
16:     __syncthreads()
17:     for $i = 1, \dots, T$ do
18:         $\mathbf{x}_i^{(t)} \leftarrow$ load_data_from_sm($\mathcal{M}^{(\tau)}$)
19:         $v_i^{(t)} \leftarrow$ load_multiplier_from_sm($\mathcal{V}^{(\tau)}$)
20:         if $\|\mathbf{x}_{\texttt{tid}}^{(t)} - \mathbf{x}_i^{(t)}\|^2 \leq R$ then
21:             $\boldsymbol{\tau} = \boldsymbol{\tau} + K_\sigma(\mathbf{x}_{\texttt{tid}}^{(t)} - \mathbf{x}_i^{(t)})\mathbf{x}_i^{(t)}$
22:             $\eta = \eta + K_\sigma(\mathbf{x}_{\texttt{tid}}^{(t)} - \mathbf{x}_i^{(t)})v_i^{(t)}$
23:         end if
24:     end for
25:     __syncthreads()
26: end for
27: if tid < N then
28:     $\mathbf{x}_{\texttt{tid}}^{(t+1)} = \boldsymbol{\tau}/\eta$
29: end if
```

5. Performance Analysis

In order to understand if these two parallel implementations (OpenMP and CUDA) could improve the performances with respect to the sequential version, a simple speedup analysis has been carried out. This analysis has been performed on synthetically-generated datasets of variable size and dimensionality. In order to avoid the curse of dimensionality, only datasets in $\mathbb{R}^2$ and $\mathbb{R}^3$ have been generated. Specifically, for each $D \in \{2, 3\}$, many datasets have been built around 3 centroids, each of different size N. This analysis involves the ones spanning from $N = 500$ to $N = 5000$. The generation has been done via Scikit-Learn's API. One example can be seen in Figure 1.

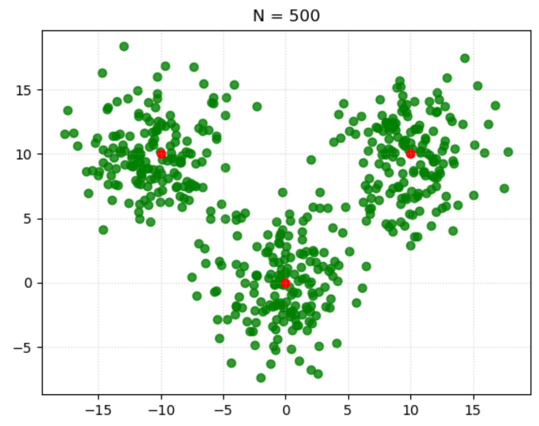

Figure 1. Example of a dataset: $N = 500, D = 2$.

For each one of the datasets involved, both the sequential and the parallel executions have been timed 10 times, in order to obtain a better estimate. For each dataset of size N and dimensionality D, the average running times for the sequential and parallel versions, $\bar{t}_s^{(N,D)}$ and $\bar{t}_p^{(N,D)}$ are obtained as:

$$\begin{cases} \bar{t}_s^{(N,D)} = \frac{1}{10} \sum_{i=1}^{10} t_s^{(i,N,D)}, \\ \bar{t}_p^{(N,D)} = \frac{1}{10} \sum_{i=1}^{10} t_p^{(i,N,D)} \end{cases} \tag{5}$$

where $t_*^{(i,N,D)}$ represents a single run. The speedup is then obtained as:

$$S^{(N,D)} = \frac{\bar{t}_s^{(N,D)}}{\bar{t}_p^{(N,D)}} \tag{6}$$

This calculation has been performed for various cases, varying the number of threads, for both CPU and GPU. In all the following plots, error bars are the standard deviations obtained from the variances propagation of Equation 6 (see Appendix A).

For bench-marking purposes no ϵ-convergence strategy has been employed: every run is composed of $I = 50$ iterations.

The CPU used for this analysis is a 4 cores / 8 threads Intel i7-860, running at 2.8GHz. The operating system was Ubuntu/Linux 5.0. GCC version was 9.3.0-17. The GPU used for this analysis is a NVIDIA GTX 980 with 4GB of dedicated memory, 2048 CUDA cores and a computing capability of 5.2. CUDA version was 11.

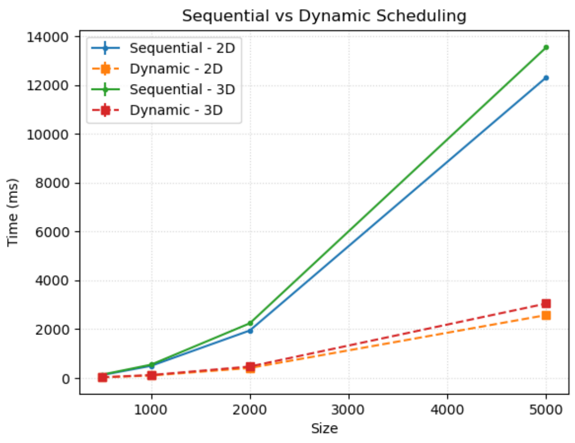

Figure 2. Execution time (*ms*) vs N: Sequential algorithm and OpenMP dynamic scheduling

All the timings were obtained while maintaining the lowest possible load average on the machine.

5.1. OpenMP

The comparison against OpenMP has been done in two ways, using both dynamic and static scheduling. The speedups have been computed against the sequential version of the algorithm and against the $T = 1$ case (*i.e.* when using OpenMP with static scheduling and 1 thread). The overall behaviour can be seen in Figure 2, where the execution time for both the sequential and dynamically scheduled OpenMP versions are shown, for each dataset considered.

The quadratic trend is clearly visible in both cases, with the parallel version being significantly faster.

Varying the number of threads, for both $D = \{2, 3\}$ a sensible speedup is reached, as can be seen in Figures 3 and 4.

For both $D = \{2, 3\}$ a maximum speedup larger than 4 has been obtained for each dataset at $T = 8$. The worse speedup $(S < 1)$ has instead been obtained for $T = 1$. It is clear, in fact, from the speedup plots (Figures 5 and 6) against this case, that $T = 1$ is worse than the sequential version, as it was expected.

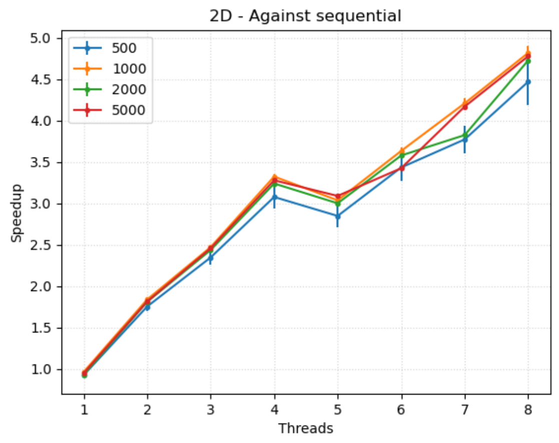

Figure 3. Speedup for $D = 2$.

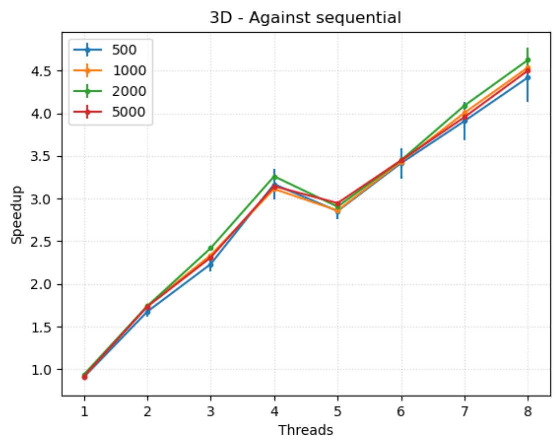

Figure 4. Speedup for $D = 3$

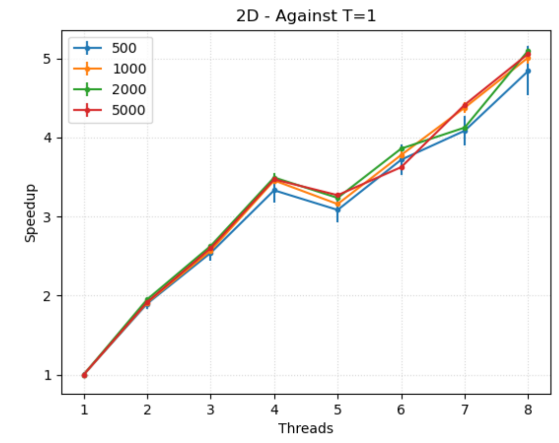

Figure 5. Speedup against $T = 1$ for $D = 2$

Finally, Figures 7 and 8 show that dynamic

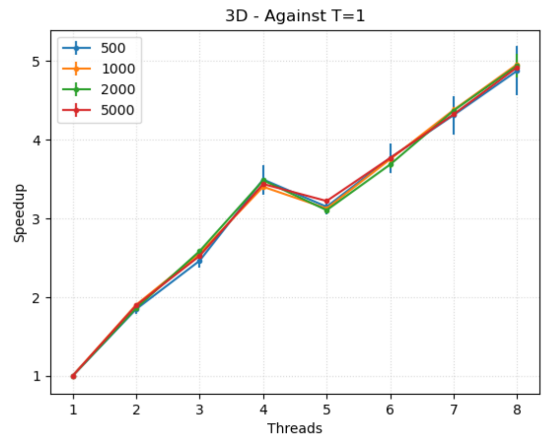

Figure 6. Speedup against $T = 1$ for $D = 3$

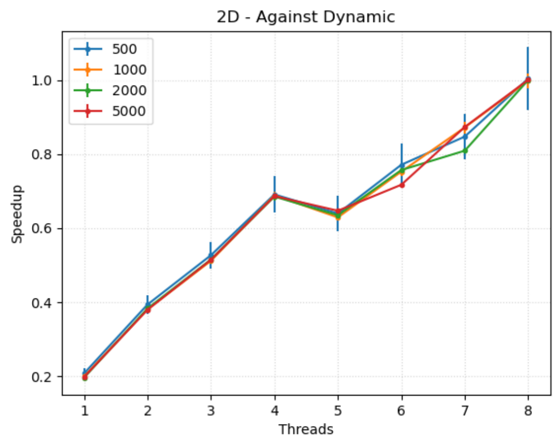

Figure 7. Speedup against dynamic scheduling for $D = 2$

scheduling behaves as the case of $T = 8$. This result, drawn from this simple performance analysis in a perfectly controlled environment, suggests that the dynamic scheduling should be the default strategy.

Interestingly when $T = 5$ there is a sensible performance drop, this may be due to the fact the benchmarking CPU has 4 physical cores and hyper-threading enabled. The empirically obtained speedup is almost linear until $T = 4$ and degrades after $T = 5$, being unable to maintain the trend. This may be explainable by the fact that CPU introduces the overhead of the hyper-threading technology management [5].

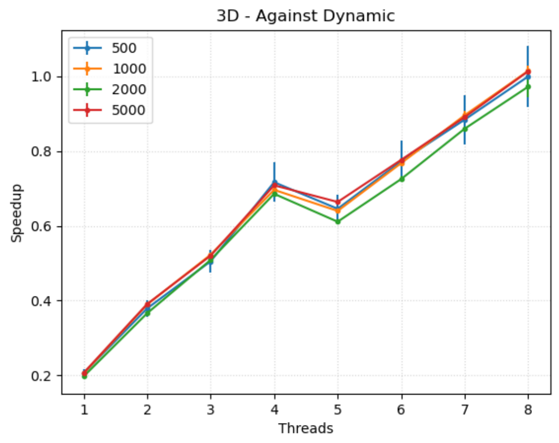

Figure 8. Speedup against dynamic scheduling for $D = 2$

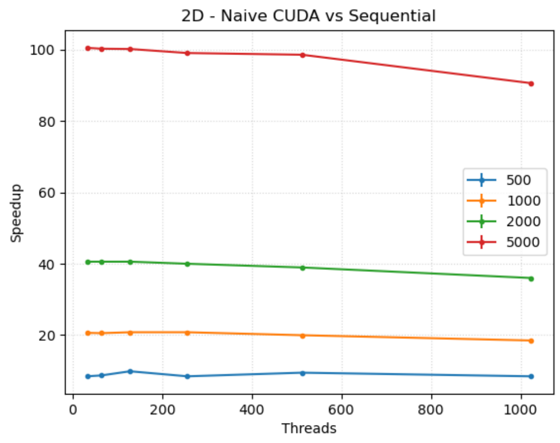

Figure 10. Speedup: CUDA naive vs sequential for $D = 2$

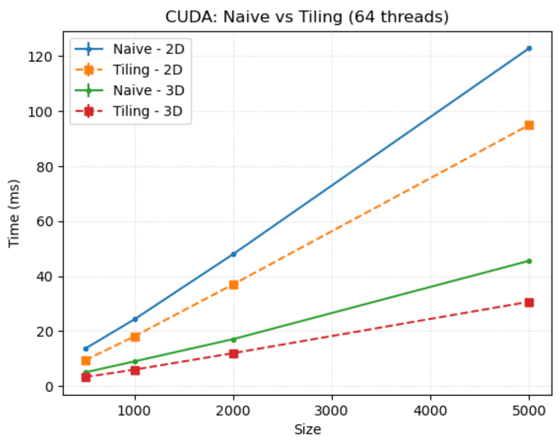

Figure 9. Naive vs tiled execution time (*ms*) for $T = 64$

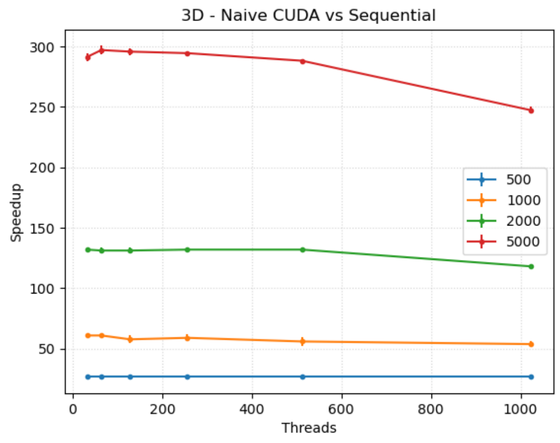

Figure 11. Speedup: CUDA naive vs sequential for $D = 3$

5.2. CUDA

The comparison against CUDA has been done confronting the sequential version against both the naive and tiled implementations. An interesting first result can be drawn from the analysis of the execution time of these two implementations, in Figure 9. For each dataset the execution time of the algorithm has been timed using $T = 64$ threads in a block.

All the four cases considered show a linear trend and, interestingly, the worst execution times are obtained for the naive version in $D = 2$, while the best ones for the tiling version in $D = 3$. This may be due to the fact that the increased workload keeps the GPU busy, making it running at 100% for the whole kernel execution time.

Figures 10 and 11 show the comparison between CUDA naive and the sequential version of the code for $D = \{2, 3\}$. This comparison is shown varying the size of $T \in \{32, 64, 128, 256, 512, 1024\}$, where T here represents the number of threads in a block.

In both cases the greatest speedups are reached for $S = 5000$, *i.e.* for the largest dataset considered. Interestingly the speedups for the case $D = 3$ are approximately 3 times greater then the ones in $D = 2$, confirming the suggestion that a larger workload could lead to a larger speedup.

Figures 12 and 13 show this same comparison strategy between CUDA tiling and the sequential version of the code for $D = \{2, 3\}$.

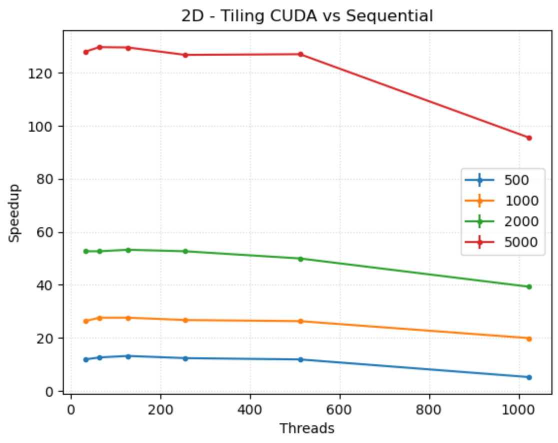

Figure 12. Speedup: CUDA tiling vs sequential for $D = 2$

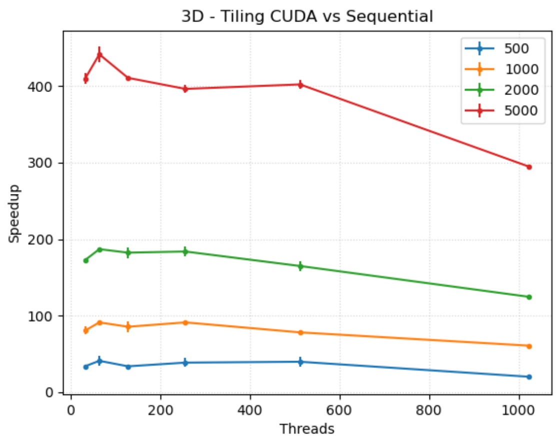

Figure 13. Speedup: CUDA tiling vs sequential for $D = 3$

Here the speedups are even greater, reaching a maximum for $D = 3$ at $S = 5000$ and $T = 64$ of, approximately, 450 times.

Overall the best performances were reached for $T \in \{32, 64, 128\}$ and the worst for $T = 1024$, being the maximum possible size for this GPU.

The trend of these speedup curves suggest that, when dealing with even larger datasets ($N > 10^5$), the user could expect an even larger speedup, as stated above.

References

[1] D. Comaniciu and P. Meer. Mean shift: a robust approach toward feature space analysis. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(5):603–619, 2002.

[2] L. Dagum and R. Menon. Openmp: An industry-standard api for shared-memory programming. 5(1):46–55, Jan. 1998.

[3] K. Fukunaga and L. Hostetler. The estimation of the gradient of a density function, with applications in pattern recognition. *IEEE Transactions on Information Theory*, 21(1):32–40, 1975.

[4] NVIDIA, P. Vingelmann, and F. H. Fitzek. Cuda, release: 11, 2020.

[5] N. H. Qun, Z. I. A. Khalib, M. N. Warip, M. E. Elobaid, R. Mostafijur, N. A. H. Zahri, and P. Saad. Hyper-threading technology: Not a good choice for speeding up cpu-bound code. In *2016 3rd International Conference on Electronic Design (ICED)*, pages 578–581, 2016.

A. Variance calculation

Dropping the indices (D, N), let $\bar{t}_s$ and $\bar{t}_p$ the average execution times for the sequential and parallel version, respectively. Let σ_s^2 and σ_p^2 be their sample variances (over 10 runs). The speedup is computed, as written in Equation 6, as

$$S = \frac{\bar{t}_s}{\bar{t}_p}$$

Variance σ^2 is then estimated via

$$\sigma^2 \approx S^2 \left[\frac{\sigma_s^2}{\bar{t}_s^2} + \frac{\sigma_p^2}{\bar{t}_p^2} \right]$$

A.1. Sanitizers

Running Cppcheck 2.3[1] should not display errors nor warnings.

Running Valgrind 3.16.1[2] should not display errors nor warnings.

Running CUDA-MEMCHECK 11.2.67[3] should not display errors nor warnings.

[1]http://cppcheck.sourceforge.net/

[2]https://valgrind.org/

[3]https://docs.nvidia.com/cuda/cuda-memcheck/index.html

B. CUDA: naive

```
__global__ void mean_shift(float *data, float *data_next) {
    size_t tid = (blockIdx.x * blockDim.x) + threadIdx.x;
    if (tid < N) {
        size_t row = tid * D;
        float new_position[D] = {0.};
        float tot_weight = 0.;
        for (size_t i = 0; i < N; ++i) {
            size_t row_n = i * D;
            float sq_dist = 0.;
            for (size_t j = 0; j < D; ++j) {
                sq_dist += (data[row + j] - data[row_n + j]) * (data[row + j] - data[row_n + j]);
            }
            if (sq_dist <= RADIUS) {
                float weight = expf(-sq_dist / DBL_SIGMA_SQ);
                for (size_t j = 0; j < D; ++j) {
                    new_position[j] += weight * data[row_n + j];
                }
                tot_weight += weight;
            }
        }
        for (size_t j = 0; j < D; ++j) {
            data_next[row + j] = new_position[j] / tot_weight;
        }
    }
    return;
}
```

C. CUDA: shared memory

```
__global__ void mean_shift_tiling(const float* data, float* data_next) {
    // Shared memory allocation
    __shared__ float local_data[TILE_WIDTH * D];
    __shared__ float valid_data[TILE_WIDTH];
    // A few convenient variables
    int tid = (blockIdx.x * blockDim.x) + threadIdx.x;
    int row = tid * D;
    int local_row = threadIdx.x * D;
    float new_position[D] = {0.};
    float tot_weight = 0.;
    // Load data in shared memory
    for (int t = 0; t < BLOCKS; ++t) {
        int tid_in_tile = t * TILE_WIDTH + threadIdx.x;
        if (tid_in_tile < N) {
            int row_in_tile = tid_in_tile * D;
            for (int j = 0; j < D; ++j) {
                local_data[local_row + j] = data[row_in_tile + j];
            }
            valid_data[threadIdx.x] = 1;
        }
        else {
            for (int j = 0; j < D; ++j) {
                local_data[local_row + j] = 0;
                valid_data[threadIdx.x] = 0;
            }
        }
        __syncthreads();
        for (int i = 0; i < TILE_WIDTH; ++i) {
            int local_row_tile = i * D;
            float valid_radius = RADIUS * valid_data[i];
            float sq_dist = 0.;
            for (int j = 0; j < D; ++j) {
                sq_dist += (data[row + j] - local_data[local_row_tile + j]) * (data[row + j] - local_data[local_row_tile + j]);
            }
            if (sq_dist <= valid_radius) {
                float weight = expf(-sq_dist / DBL_SIGMA_SQ);
                for (int j = 0; j < D; ++j) {
                    new_position[j] += (weight * local_data[local_row_tile + j]);
                }
                tot_weight += (weight * valid_data[i]);
            }
        }
        __syncthreads();
    }
    if (tid < N) {
        for (int j = 0; j < D; ++j) {
            data_next[row + j] = new_position[j] / tot_weight;
        }
    }
    return;
}
```